

REMARKS

I. Summary of Amendments

By this amendment, Applicants propose amending claims 55, 66, 67, and 69–73. Claims 55–74 are pending.

II. Summary of Office Action

In the last Office Action, the Examiner

- (a) rejected claims 66–73 under 35 U.S.C. § 112,
- (b) rejected claims 55–64 under 35 U.S.C. § 101,
- (c) rejected claims 55–58, 64–68, and 74 under 35 U.S.C. § 102(e) over “Java Virtual Machine Profiler Interface (JVMPI),” November 11, 1998, and
- (d) rejected claims 59–63 and 69–73 under 35 U.S.C. § 103(a) over JVMPI and U.S. Patent No. 5,857,210 (Tremblay).

III. Rejection under 35 U.S.C. § 112

Applicants propose amending claims 66, 67, and 69–73, as shown above in the listing of claims, to correct a typographical error. In view of those proposed amendments, Applicants request withdrawal of the rejection of claims 66, 67, and 69–73 under 35 U.S.C. § 112.

IV. Rejection under 35 U.S.C. § 101

Applicants traverse the rejection of claims 55–64 under 35 U.S.C. § 101. The Examiner on page 3 of the Office Action alleges that claim 55 does not appear to contain elements related to a device or physical apparatus. Applicants respectfully

disagree. In order to advance the prosecution of this application, however, Applicants have proposed amending independent claim 55 to recite “a processor” and “memory,” as shown above in the listing of claims. Accordingly, the rejection of claim 55 and its dependent claims 56–64 under 35 U.S.C. § 101 should be withdrawn.

V. Rejection under 35 U.S.C. § 102(e)

Applicants traverse the rejection of claims 55–58, 64–68, and 74 under 35 U.S.C. § 102(e) over JVMPI.

Applicants previously submitted a copy of JVMPI as Appendix A of the specification. Further, in patent application no. 09/858,779, which is the parent application for the present application, Applicants filed Declarations Under 37 C.F.R. § 1.132 (copies attached), on October 23, 2006 and June 19, 2007, verifying that they were the authors of the subject matter in JVMPI. Thus, pursuant to 37 C.F.R. § 716.10, JVMPI does not qualify as prior art under 35 U.S.C. § 102(e). Accordingly, the rejection of claims 55–58, 64–68, and 74 under 35 U.S.C. § 102(e) should be withdrawn.

VI. Rejection under 35 U.S.C. § 103(a)

Applicants traverse the rejection of claims 59–63 and 69–73 under 35 U.S.C. § 103(a) over JVMPI and Tremblay.

As discussed above, the primary reference, JVMPI, does not qualify as prior art because the Applicants are the authors of the subject matter in JVMPI. Thus, the Examiner has not established a prima facie case of obviousness using JVMPI and Tremblay. Accordingly, the rejection of claims 59–63 and 69–73 under 35 U.S.C. § 103(a) over those references should be withdrawn.

VII. Conclusion

Applicants respectfully request that this Amendment Under 37 C.F.R. § 1.116 be entered by the Examiner because it should allow for immediate allowance of the pending claims by the Examiner. Further, Applicants submit that the entry of the amendment would place the application in better form for appeal, should the Examiner dispute the patentability of the pending claims.

If the Examiner wishes to discuss any aspect of this application, Applicants invite the Examiner to contact Applicants' representative undersigned below.

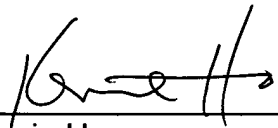
Please grant any extensions of time required to enter this response and charge any additional required fees to Deposit Account 06-0916.

Respectfully submitted,

FINNEGAN, HENDERSON, FARABOW,
GARRETT & DUNNER, L.L.P.

Dated: June 30, 2008

By: _____


Kenie Ho
Reg. No. 51,808
(202) 408-4287

Attachments:

Declaration Under 37 C.F.R. § 1.132 of Sheng Liang

Declaration Under 37 C.F.R. § 1.132 of Steffen Grarup



PATENT
Attorney Docket No. 06502.0523-00

THE UNITED STATES PATENT AND TRADEMARK OFFICE



In re Application of:

SHENG LIANG et al.

Application No.: 09/856,779

Filed: October 3, 2001

For: A METHOD FOR ENABLING
COMPREHENSIVE PROFILING OF
GARBAGE-COLLECTED MEMORY
SYSTEMS

)
)
) Group Art Unit: 2187
)
) Examiner: Brian R. Peugh
)
) Confirmation No.: 3704
)

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Sir:

DECLARATION UNDER 37 C.F.R. § 1.132

I, Sheng Liang, do hereby make the following declaration:

1. I authored the document, "Java Virtual Machine Profiler Interface (JVMPi)," November 11, 1998 (attached as Appendix A).
2. To the extent that Appendix A describes the invention(s) in the above-referenced patent application, Steffen Grarup and I are the inventors.

I further declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true, and further, that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application or any patent issuing thereon.

Dated: 10/10/06

By: 
Sheng Liang

Java Virtual Machine Profiler Interface (JVMPi)

This document describes the Java Virtual Machine Profiler Interface (JVMPi) in JDK 1.2. It is intended for tools vendors to develop profilers that work in conjunction with Sun's Java virtual machine implementation.

Contents

- Overview
 - Start-up
 - Function Call Interface
 - Event Notification
 - JVMPi IDs
 - Threading and Locking Issues
 - Data Communication between the Profiler Agent and Front-End
- Interface Functions
 - CreateSystemThread
 - DisableEvent
 - DisableGC
 - EnableEvent
 - EnableGC
 - GetCallTrace
 - GetCurrentThreadCpuTime
 - GetMethodClass
 - GetThreadLocalStorage
 - GetThreadObject
 - GetThreadStatus
 - NotifyEvent
 - ProfilerExit
 - RawMonitorCreate
 - RawMonitorDestroy
 - RawMonitorEnter
 - RawMonitorExit
 - RawMonitorNotifyAll
 - RawMonitorWait
 - RequestEvent
 - ResumeThread
 - RunGC
 - SetThreadLocalStorage
 - SuspendThread
 - ThreadHasRun
- Events
 - JVMPi EVENT ARENA DELETE
 - JVMPi EVENT ARENA NEW
 - JVMPi EVENT CLASS LOAD
 - JVMPi EVENT CLASS LOAD HOOK
 - JVMPi EVENT CLASS UNLOAD

- JVMPI EVENT COMPILED METHOD LOAD
 - JVMPI EVENT COMPILED METHOD UNLOAD
 - JVMPI EVENT DATA DUMP REQUEST
 - JVMPI EVENT DATA RESET REQUEST
 - JVMPI EVENT GC FINISH
 - JVMPI EVENT GC START
 - JVMPI EVENT HEAP DUMP
 - JVMPI EVENT JNI GLOBALREF ALLOC
 - JVMPI EVENT JNI GLOBALREF FREE
 - JVMPI EVENT JNI WEAK GLOBALREF ALLOC
 - JVMPI EVENT JNI WEAK GLOBALREF FREE
 - JVMPI EVENT JVM INIT DONE
 - JVMPI EVENT JVM SHUT DOWN
 - JVMPI EVENT METHOD ENTRY
 - JVMPI EVENT METHOD ENTRY2
 - JVMPI EVENT METHOD EXIT
 - JVMPI EVENT MONITOR CONTENTED ENTER
 - JVMPI EVENT MONITOR CONTENTED ENTERED
 - JVMPI EVENT MONITOR CONTENTED EXIT
 - JVMPI EVENT MONITOR DUMP
 - JVMPI EVENT MONITOR WAIT
 - JVMPI EVENT MONITOR WAITED
 - JVMPI EVENT OBJECT ALLOC
 - JVMPI EVENT OBJECT DUMP
 - JVMPI EVENT OBJECT FREE
 - JVMPI EVENT OBJECT MOVE
 - JVMPI EVENT RAW MONITOR CONTENTED ENTER
 - JVMPI EVENT RAW MONITOR CONTENTED ENTERED
 - JVMPI EVENT RAW MONITOR CONTENTED EXIT
 - JVMPI EVENT THREAD END
 - JVMPI EVENT THREAD START
 - Dump Formats
 - Sizes and Types Used in Dump Format Descriptions
 - Heap Dump Format
 - Object Dump Format
 - Monitor Dump Format
 - Data Types
 - objectID
 - JVMPI CallFrame
 - JVMPI CallTrace
 - JVMPI Field
 - JVMPI HeapDumpArg
 - JVMPI Lineno
 - JVMPI Method
 - JVMPI RawMonitor
 - Notes on JDK1.2 Implementation Limitations
-

1. Overview

The JVMPI is a two-way function call interface between the Java virtual machine and an in-process profiler agent. On one hand, the virtual machine notifies the profiler agent of various events, corresponding to, for example, heap allocation, thread start, etc. On the other hand, the profiler agent issues controls and requests for more information through the JVMPI. For example, the profiler agent can turn on/off a specific event notification, based on the needs of the profiler front-end.



The profiler front-end may or may not run in the same process as the profiler agent. It may reside in a different process on the same machine, or on a remote machine connected via the network. The JVMPI does not specify a standard wire protocol. Tools vendors may design wire protocols suitable for the needs of different profiler front-ends.

A profiling tool based on JVMPI can obtain a variety of information such as heavy memory allocation sites, CPU usage hot-spots, unnecessary object retention, and monitor contention, for a comprehensive performance analysis.

JVMPI supports partial profiling, i.e. a user can selectively profile an application for certain subsets of the time the virtual machine is up and can also choose to obtain only certain types of profiling information.

In the current version of JVMPI, only one agent per virtual machine can be supported.

1.1. Start-up

The user can specify the name of the profiler agent and the options to the profiler agent through a command line option to the Java virtual machine. For example, suppose the user specifies:

```
java -Xrunmyprofiler:heapdump=on,file=log.txt ToBeProfiledClass
```

The VM attempts to locate a profiler agent library called `myprofiler` in Java's library directory:

- On Win32, it is `$JAVA_HOME\bin\myprofiler.dll`
- On SPARC/Solaris, it is `$JAVA_HOME/lib/sparc/libmyprofiler.so`

If the library is not found in the Java library directory, the VM continues to search for the library following the normal library search mechanism of the given platform:

- On Win32, the VM searches the current directory, Windows system directories, and the directories in the `PATH` environment variable.
- On Solaris, the VM searches the directories in `LD_LIBRARY_PATH`.

The VM loads the profiler agent library and looks for the entry point:

```
jint JNICALL JVM_OnLoad(JavaVM *jvm, char *options, void *reserved);
```

The VM calls the `JVM_OnLoad` function, passing a pointer to the `JavaVM` instance as the first argument, and string `"heapdump=on,file=log.txt"` as the second argument. The third argument to `JVM_OnLoad` is reserved and set to `NULL`.

On success, the `JVM_OnLoad` function must return `JNI_OK`. If for some reason the `JVM_OnLoad` function fails, it must return `JNI_ERR`.

1.2. Function Call Interface

The profiler agent can obtain a function call interface by issuing a `GetEnv` call on the `JavaVM` pointer. For example, the following code retrieves the version of JVMPI interface that is implemented in JDK

1.2:

```

JVMPI_Interface *jvm_pi_interface;

JNIEXPORT jint JNICALL JVM_OnLoad(JavaVM *jvm, char *options, void *reserved)
{
    jint res = (*jvm)->GetEnv(jvm, (void **) &jvm_pi_interface, JVMPI_VERSION_1)
    if (res < 0) {
        return JNI_ERR;
    }
    ... /* use entries in jvm_pi_interface */
}

```

The JVMPI_Interface structure defines the function call interface between the profiler agent and the VM:

```

/* interface functions */
typedef struct {
    jint version; /* JVMPI version */

    /* -----interface implemented by the profiler----- */
    void (*NotifyEvent)(JVMPI_Event *event);

    /* -----interface implemented by the JVM----- */

    jint (*EnableEvent)(jint event_type, void *arg);
    jint (*DisableEvent)(jint event_type, void *arg);
    jint (*RequestEvent)(jint event_type, void *arg);

    void (*GetCallTrace)(JVMPI_CallTrace *trace, jint depth);

    void (*ProfilerExit)(jint);

    JVMPI_RawMonitor (*RawMonitorCreate)(char *lock_name);
    void (*RawMonitorEnter)(JVMPI_RawMonitor lock_id);
    void (*RawMonitorExit)(JVMPI_RawMonitor lock_id);
    void (*RawMonitorWait)(JVMPI_RawMonitor lock_id, jlong ms);
    void (*RawMonitorNotifyAll)(JVMPI_RawMonitor lock_id);
    void (*RawMonitorDestroy)(JVMPI_RawMonitor lock_id);

    jlong (*GetCurrentThreadCpuTime)(void);
    void (*SuspendThread)(JNIEnv *env);
    void (*ResumeThread)(JNIEnv *env);
    jint (*GetThreadStatus)(JNIEnv *env);
    jboolean (*ThreadHasRun)(JNIEnv *env);
    jint (*CreateSystemThread)(char *name, jint priority, void (*f)(void *));
    void (*SetThreadLocalStorage)(JNIEnv *env_id, void *ptr);
    void * (*GetThreadLocalStorage)(JNIEnv *env_id);

    void (*DisableGC)(void);
    void (*EnableGC)(void);
    void (*RunGC)(void);

    jobjectID (*GetThreadObject)(JNIEnv *env);
    jobjectID (*GetMethodClass)(jmethodID mid);
} JVMPI_Interface;

```

The GetEnv function returns a pointer to a JVMPI_Interface whose version field indicates a JVMPI version that is compatible to the version number argument passed in the GetEnv call. Note that the value of the version field is not necessarily identical to the version argument passed in the

GetEnv call.

The `JVMPInterface` returned by `GetEnv` has all the functions set up except for `NotifyEvent`. The profiler agent must set up the `NotifyEvent` function pointer before returning from `JVM_OnLoad`.

1.3. Event Notification

The VM sends an event by calling `NotifyEvent` with a `JVMPEvent` data structure as the argument. The following events are supported:

- method enter and exit
- object alloc, move, and free
- heap arena create and delete
- GC start and finish
- JNI global reference alloc and free
- JNI weak global reference alloc and free
- compiled method load and unload
- thread start and end
- class file data ready for instrumentation
- class load and unload
- contended Java monitor wait to enter, entered, and exit
- contended raw monitor wait to enter, entered, and exit
- Java monitor wait and waited
- monitor dump
- heap dump
- object dump
- request to dump or reset profiling data
- Java virtual machine initialization and shutdown

The `JVMPEvent` structure contains the event type, the `JNIEnv` pointer of the current thread, and other event-specific information. The event specific information is represented as a union of event-specific structures. The `JVMPEvents` section provides a complete description of all event-specific structures. For now, we show the event-specific structures for class load and class unload below.

```
typedef struct {
    jint event_type;           /* event_type */
    JNIEnv *env_id;          /* env where this event occurred */

    union {
        struct {
            char *class_name; /* class name */
            char *source_name; /* name of source file */
            jint num_interfaces; /* number of interfaces implemented */
            jint num_methods; /* number of methods in the class */
            JVMPI_Method *methods; /* methods */
            jint num_static_fields; /* number of static fields */
            JVMPI_Field *statics; /* static fields */
            jint num_instance_fields; /* number of instance fields */
            JVMPI_Field *instances; /* instance fields */
            jobjectID class_id; /* id of the class object */
        } class_load;

        struct {
            jobjectID class_id; /* id of the class object */
        } class_unload;

        ... /* Refer to the section on JVMPI events for a full listing */
    } u;
};
```

| JVMPI_Event:

1.4. JVMPI IDs

The JVMPI refers to entities in the Java virtual machine as various kinds of IDs. Threads, classes, methods, objects, heap arenas and JNI global references all have unique IDs.

Each ID has a defining event and an undefining event. A defining event provides the information related to an ID. For example, the defining event for a thread ID contains, among other entries, the name of the thread.

An ID is valid until its undefining event arrives. An undefining event invalidates the ID, whose value may be reused later as a different kind of ID. The value of a thread ID, for example, may be redefined as a method ID after the thread ends.

ID	data type	defining event	undefining event
thread ID	JNIEnv *	thread start	thread end
object ID	jobjectID	object alloc	object free, object move, and arena delete
class ID	jobjectID	class load	class unload and object move
method ID	jmethodID	defining class load	defining class unload
arena ID	jint	arena new	arena delete
JNI global ref ID	jobject	global ref alloc	global ref free

Assuming the defining events are enabled during the profiler initialization, the profiler agent is guaranteed to be notified of an entity's creation through a defining event, before the entity appears in other JVMPI events.

If the defining events are not enabled, the profiler agent may receive an unknown ID. In that case the profiler agent may request the corresponding defining event to be sent on demand by issuing a RequestEvent call.

IDs representing objects have type jobjectID. A class is represented by the object ID of the corresponding `java.lang.Class` object. Therefore, class IDs are also of type jobjectID.

A jobjectID is defined by an object alloc event, and remains valid in the arena in which the object is allocated until one of its undefining events arrive:

- An object free event invalidates an object ID.
- An object move event is a special type of undefining events. Unlike other undefining events which signal the end-of-life of the corresponding entities, the object still exists, but its ID changes, and it may have been moved to a new arena.
- An arena delete event invalidates all remaining object IDs in the arena.

When an object free or arena delete event invalidates an object ID, the object is known as being garbage collected.

Typically, the profiler agent maintains a mapping between jobjectIDs and its internal representation of object identities, and updates the mapping in response to the defining and undefining events for JVMPI object IDs.

Since object IDs may be invalidated during GC, the VM issues all events that contain jobjectID entries with GC disabled. In addition, the profiling agent must disable GC when it is directly

manipulating any `jobjectID` data types. Otherwise the GC may invalidate a `jobjectID` while it is being manipulated in the agent code. The profiler agent must make sure that GC is disabled when it calls a JVMPI function that either takes a `jobjectID` argument or returns a `jobjectID` result. If the function call is inside an event handler where GC is already disabled, then the profiler agent need not explicitly disable the GC again.

A thread may be identified either by its `JNIEnv` interface pointer or by the object ID of the corresponding `java.lang.Thread` object. The `JNIEnv` pointer is valid between thread start and thread end events, and remains constant during the lifetime of a thread. The `java.lang.Thread` object ID, on the other hand, could remain valid after the thread ends, until it is garbage collected. The profiler agent can convert a `JNIEnv` pointer to the corresponding thread object ID by calling the `GetThreadObject` function.

1.5. Threading and Locking Issues

The JVMPI is used by the profiler agent that runs in the same process as the Java virtual machine. Programmers who write the agent must be careful in dealing with threading and locking issues in order to prevent data corruption and deadlocks.

Events are sent in the same thread where they are generated. For example, a class loading event is sent in the same thread in which the class is loaded. Multiple events may arrive concurrently in different threads. The agent program must therefore provide the necessary synchronization in order to avoid data corruption caused by multiple threads updating the same data structure at the same time.

In some cases, synchronizing on certain frequent events (such as method entry and method exit) may impose unacceptable overhead to program execution. Agents may utilize the thread-local storage support provided by the JVMPI to record profiling data without having to contend for global locks, and only merge the thread-local data into global profiles at selected intervals. The JVMPI supplies the agent with a pointer-size thread-local storage. Following is a simple example that illustrates how a profiler agent may take advantage of this feature. Suppose we need to write a profiler agent that counts the number of methods executed in each thread. The agent installs event handlers for thread start, method entry, and thread end events:

```
/* thread start event handler
 * sets up the storage for thread-local method invocation counter
 */
void ThreadStartHandler(JNIEnv *thread_id)
{
    int *p_ctr = (int *)malloc(sizeof(int));
    CALL(SetThreadLocalStorage)(thread_id, p_ctr);
}

/* method enter event handler
 * increments thread local method invocation counter
 */
void MethodEntryHandler(jmethodID method_id, JNIEnv *thread_id)
{
    int *p_ctr = (int *)CALL(GetThreadLocalStorage)(thread_id);
    (*p_ctr)++;
}

/* thread end handler
 * prints the number of methods executed
 */
void ThreadEndHandler(JNIEnv *thread_id)
{
    int *p_ctr = (int *)CALL(GetThreadLocalStorage)(thread_id);
    printf(stdout, "Thread %x executed %d methods\n",
```

```

        thread_id. (*p_ctr));
    free(p_ctr);
}

```

The following JVMPI functions can cause event notification to be sent synchronously in the same thread during the function execution:

- RequestEvent
- CreateSystemThread
- RunGC

The `RequestEvent` function supplies the JVMPI event explicitly requested by the profiler agent. The `CreateSystemThread` function causes thread object allocation and thread start events to be issued. The `RunGC` function causes GC-related events to be generated.

When a profiling agent is loaded into the Java virtual machine, the process can either be in one of three modes: multi-threaded mode with GC enabled, multi-threaded mode with GC disabled, and the thread suspended mode. Different JVMPI events are issued in different modes. Certain JVMPI functions change the process from one mode to another.

The profiler agent must obey the following guidelines to avoid deadlocks:

- In the multi-threaded mode with GC enabled, the agent code has a great deal of freedom in acquiring locks and calling JVMPI functions. Of course the normal rules of deadlock avoidance apply. Different threads must not enter the same set of locks in different orders.
- When the GC is disabled the agent program must not call any JVMPI function that could require new Java objects to be created or cause the garbage collector to run. Currently, such functions include `CreateSystemThread` and `RunGC`. In addition, programmers need to be aware that disabling the GC creates an implicit locking dependency among threads. When the GC is disabled, the current thread may not be able to safely acquire certain locks. Deadlocks may happen, for example, if one thread disables GC and tries to acquire a lock, while another thread already acquired that lock but is triggering a GC.
- In the thread suspended mode, one or more of the threads have been suspended. In this case, the agent program must not perform any operations that may cause the current thread to block. Such operations include, for example, the `malloc` and `fprintf` functions provided by the standard C library. These functions typically acquire internal C library locks that may be held by one of the suspended threads.

1.6 Data Communication between the Profiler Agent and Front-End

The JVMPI provides a low-level mechanism for a profiler agent to communicate with the virtual machine. The goal is to provide maximum flexibility for the profiler agent to present the data depending on the needs of the front-end, and also to keep the processing work done by the virtual machine at a minimum. Therefore, the JVMPI does not specify a wire protocol between the profiling agent and the front-end. Instead, tools vendors design their own profiling agents that suit the needs of their front-ends.

The following issues need to be considered when designing the wire protocol in order to allow the profiler agent and front-end to reside on different machines:

- Pointer size (e.g., 32 or 64 bit) - all of the JVMPI IDs are of pointer type (see Data Types).
- Byte order (little endian or big endian).
- Bit order (most significant bit first or least significant bit first).

- String encoding - the JVMPI uses the UTF-8 encoding as documented in the Java virtual machine specification.

For example, the *hprof* profiler agent shipped with JDK 1.2 sends the size of all IDs as the first record, and uses the standard network byte order for integer and floating-point data.

2. Interface Functions

```
jint (*CreateSystemThread)(char *name, jint priority, void (*f)(void *));
```

Called by the profiler agent to create a daemon thread in the Java virtual machine.

It is safe for the profiler agent to make this call only after the JVM notifies a `JVMPI_EVENT_INIT_DONE` and when the system is in a multi-threaded mode with GC enabled.

Arguments:

`name` - name of the thread.
`priority` - thread priority; the values can be:
`JVMPI_NORMAL_PRIORITY`
`JVMPI_MAXIMUM_PRIORITY`
`JVMPI_MINIMUM_PRIORITY`
`f` - function to be run by the thread.

Returns:

`JNI_OK` - success.
`JNI_ERR` - failure.

```
jint (*DisableEvent)(jint event_type, void *arg);
```

Called by the profiler agent to disable the notification of a particular type of event. Apart from `event_type`, the profiler agent may also pass an argument that provides additional information specific to the given event type.

All events are disabled when the VM starts up. Once enabled, an event stays enabled until it is explicitly disabled.

This function returns `JVMPI_NOT_AVAILABLE` if `event_type` is `JVMPI_EVENT_HEAP_DUMP`, `JVMPI_EVENT_MONITOR_DUMP` or `JVMPI_EVENT_OBJECT_DUMP`.

Arguments:

`event_type` - type of event, `JVMPI_EVENT_CLASS_LOAD` etc.
`arg` - event specific information.

Returns:

JVMPI_SUCCESS	disable succeeded.
JVMPI_FAIL	disable failed.
JVMPI_NOT_AVAILABLE	support for disabling the given event_type is not available.

void (*DisableGC) (void);

Called by the profiler to disable garbage collection, until `EnabledGC` is called. `DisableGC` and `EnabledGC` calls may be nested.

jint (*EnableEvent) (jint event_type, void *arg);

Called by the profiler agent to enable notification of a particular type of event. Apart from `event_type`, the profiler may also pass an argument that provides additional information specific to the given event type.

All events are disabled when the VM starts up. Once enabled, an event stays enabled until it is explicitly disabled.

This function returns `JVMPI_NOT_AVAILABLE` if `event_type` is `JVMPI_EVENT_HEAP_DUMP`, `JVMPI_EVENT_MONITOR_DUMP` or `JVMPI_EVENT_OBJECT_DUMP`. The profiler agent must use the `RequestEvent` function to request these events.

Arguments:

`event_type` - type of event, `JVMPI_EVENT_CLASS_LOAD` etc.
`arg` - event specific argument.

Returns:

JVMPI_SUCCESS	enable succeeded.
JVMPI_FAIL	enable failed.
JVMPI_NOT_AVAILABLE	support for enabling the given event_type is not available.

void (*EnableGC) (void);

Enables garbage collections. `DisableGC` and `EnableGC` calls may be nested.

void (*GetCallTrace) (JVMPI_CallTrace *trace, jint depth);

Called by the profiler to obtain the current method call stack trace for a given thread. The thread is identified by the `env_id` field in the `JVMPI_CallTrace` structure. The profiler agent should allocate a `JVMPI_CallTrace` structure with enough memory for the requested stack depth. The VM fills in the `frames` buffer and the `num_frames` field.

Arguments:

`trace` - trace data structure to be filled by the VM.
`depth` - depth of the call stack trace.

```
jlong (*GetCurrentThreadCpuTime) (void);
```

Called by the profiler agent to obtain the accumulated CPU time consumed by the current thread.

Returns:

time in nanoseconds

```
jobjectID (*GetMethodClass) (jmethodID mid);
```

Called by the profiler agent to obtain the object ID of the class that defines a method.

The profiler must disable GC before calling this function.

Arguments:

mid - a method ID.

Returns:

object ID of the defining class.

```
void * (*GetThreadLocalStorage) (JNIEnv *env_id);
```

Called by the profiler to get the value of the JVMPI thread-local storage. The JVMPI supplies to the agent a pointer-size thread-local storage that can be used to record per-thread profiling information.

Arguments:

env_id - the JNIEnv * of the thread.

Returns:

the value of the thread local storage

```
jobjectID (*GetThreadObject) (JNIEnv *env);
```

Called by the profiler agent to obtain the thread object ID that corresponds to a JNIEnv pointer.

The profiler must disable GC before calling this function.

Arguments:

env - JNIEnv pointer of the thread.

Returns:

the thread object ID.

```
jint (*GetThreadStatus) (JNIEnv *env);
```

Called by the profiler agent to obtain the status of a thread.

The JVMPI functions `SuspendThread` and `ResumeThread` have no affect on the status returned by `GetThreadStatus`. The status of a thread suspended through the JVMPI remains unchanged and the status at the time of suspension is returned.

Arguments:

`env` - the `JNIEnv` of the thread.

Returns:

`JVMPI_THREAD_RUNNABLE` - thread is runnable.

`JVMPI_THREAD_MONITOR_WAIT` - thread is waiting on a monitor.

`JVMPI_THREAD_CONDVAR_WAIT` - thread is waiting on a condition variable.

When a thread is suspended (by `java.lang.Thread.suspend`) or interrupted in any of the above states the `JVMPI_THREAD_SUSPENDED` or the `JVMPI_THREAD_INTERRUPTED` bit is set.

```
void (*NotifyEvent) (JVMPI_Event *event);
```

Called by the VM to send an event to the profiling agent. The profiler agent registers the types of events it is interested in by calling `EnableEvent`, or requests a specific type of event by calling `RequestEvent`.

When an event is enabled by `EnableEvent`, the thread that generates the event is the thread in which the event is sent. When an event is requested by `RequestEvent`, the thread that requests the event is the thread in which the event is sent. Multiple threads may send multiple events concurrently.

If the event specific information contains a `jobjectID`, this function is called with GC disabled. GC is enabled after the function returns.

The space allocated for the `JVMPI_Event` structure and any event specific information is freed by the virtual machine once this function returns. The profiler agent must copy any necessary data it needs to retain into its internal buffers.

Arguments:

`event` - the JVMPI event sent from the VM to the profiling agent.

```
void (*ProfilerExit) (jint err_code);
```

Called by the profiler agent to inform the VM that the profiler wants to exit with error code set to `err_code`. This function causes the VM to also exit with the same `err_code`.

Arguments:

`err_code` - exit code

JVMPi_RawMonitor (***RawMonitorCreate**) (`char *lock_name`);

Called by the profiler to create a raw monitor.

Raw monitors are similar to Java monitors. The difference is that raw monitors are not associated with Java objects.

It is not safe for the profiler agent to call this function in the thread suspended mode because this function may call arbitrary system functions such as `malloc` and block on an internal system library lock.

If the raw monitor is created with a name beginning with an underscore ('_'), then its monitor contention events are not sent to the profiler agent.

Arguments:

`lock_name` - name of raw monitor.

Returns:

a raw monitor

void (***RawMonitorDestroy**) (**JVMPi_RawMonitor** `lock_id`);

Called by the profiler agent to destroy a raw monitor and free all system resources associated with the monitor.

Raw monitors are similar to Java monitors. The difference is that raw monitors are not associated with Java objects.

It is not safe for the profiler agent to call this function in the thread suspended mode because this function may call arbitrary system functions such as `free` and block on a internal system library lock.

Arguments:

`lock_id` - the raw monitor to be destroyed

void (***RawMonitorEnter**) (**JVMPi_RawMonitor** `lock_id`);

Called by the profiler agent to enter a raw monitor.

Raw monitors are similar to Java monitors. The difference is that raw monitors are not associated with Java objects.

It is not safe for the profiler agent to call this function in the thread suspended mode because

the current thread may block on the raw monitor already acquired by one of the suspended threads:

Arguments:

`lock_id` - the raw monitor to be entered

```
void (*RawMonitorExit) (JVMPI_RawMonitor lock_id);
```

Called by the profiler agent to exit a raw monitor.

Raw monitors are similar to Java monitors. The difference is that raw monitors are not associated with Java objects.

Arguments:

`lock_id` - the raw monitor to exit

```
void (*RawMonitorNotifyAll) (JVMPI_RawMonitor lock_id);
```

Called by the profiler to notify all the threads that are waiting on a raw monitor.

Raw monitors are similar to Java monitors. The difference is that raw monitors are not associated with Java objects.

Arguments:

`lock_id` - the raw monitor to notify

```
void (*RawMonitorWait) (JVMPI_RawMonitor lock_id, jlong ms);
```

Called by the profiler agent to wait on a raw monitor for a specified timeout period. Passing 0 as the timeout period causes the thread to wait forever.

Raw monitors are similar to Java monitors. The difference is that raw monitors are not associated with Java objects.

Arguments:

`lock_id` - the raw monitor to wait on
`ms` - time to wait (in milliseconds).

```
jint (*RequestEvent) (jint event_type, void *arg);
```

Called by the profiler agent to request a particular type of event to be notified. Apart from `event_type`, the profiler agent may also pass an argument that provides additional information specific to the given event type.

This function can be called to request one-time events such as `JVMPI_EVENT_HEAP_DUMP`, `JVMPI_EVENT_MONITOR_DUMP` and `JVMPI_EVENT_OBJECT_DUMP`. Notification for these events cannot be controlled by the `EnableEvent` and `DisableEvent` functions.

In addition, this function can be called to request the *defining events* for a specific class, thread, or object. This is useful when the profiler agent needs to resolve an unknown class, method, thread, or object ID received in an event, but the corresponding defining event was disabled earlier.

- The profiler agent may receive information about an unknown class ID by requesting a JVMPI_EVENT_CLASS_LOAD event and setting the event-specific argument to the class object ID.
- The profiler agent may receive information about an unknown thread ID by requesting a JVMPI_EVENT_THREAD_START event and setting the event-specific argument to the thread object ID.
- The profiler agent may receive information about an unknown object ID by requesting a JVMPI_EVENT_OBJECT_ALLOC event and setting the event-specific argument to the object ID.

Thus the profiler agent can either enable the above three events asynchronously by calling EnableEvent, or request these events synchronously by calling RequestEvent. The requested event is sent in the same thread that issued the RequestEvent call, and is sent before the RequestEvent function returns.

The RequestEvent function cannot be used to request other events not listed above.

Events requested through RequestEvent will arrive with the JVMPI_REQUESTED_EVENT bit set in its event_type.

Arguments:

event_type - type of event, JVMPI_EVENT_CLASS_LOAD etc.
arg - event specific argument.

Returns:

JVMPI_SUCCESS request succeeded.
JVMPI_FAIL request failed.
JVMPI_NOT_AVAILABLE support for issuing the requested event_type is not available.

```
void (*ResumeThread) (JNIEnv *env);
```

Called by the profiler agent to resume a thread.

Note that a thread suspended by the `java.lang.Thread.suspend` method cannot be resumed by the JVMPI ResumeThread function.

Arguments:

env - the JNIEnv * of the thread.

```
void (*RunGC) (void);
```

Called by the profiler to force a complete garbage collection. This function must not be called when GC is disabled.

```
void (*SetThreadLocalStorage) (JNIEnv *env_id, void *ptr);
```

Called by the profiler agent to set the value of the JVMPI thread-local storage. The JVMPI supplies to the agent a pointer-size thread-local storage that can be used to record per-thread profiling information.

Arguments:

`env_id` - the JNIEnv * of the thread.
`ptr` - the value to be entered into the thread-local storage.

```
void (*SuspendThread) (JNIEnv *env);
```

Called by the profiler agent to suspend a thread. The system enters the thread suspended mode after this function is called.

Note that a thread suspended by the JVMPI `SuspendThread` function cannot be resumed by the `java.lang.Thread.resume` method.

In the JDK 1.2 implementation, this function must be called when the GC is disabled. GC must remain disabled until all threads have been resumed.

Arguments:

`env` - the JNIEnv * of the thread.

```
jboolean (*ThreadHasRun) (JNIEnv *env);
```

Called by the profiler to determine if a thread identified by the given JNIEnv pointer has consumed CPU time since the last time the thread was suspended by `SuspendThread`. This function must be called when the thread has been resumed by `ResumeThread` and then suspended again by the `SuspendThread` function.

Arguments:

`env` - the JNIEnv * of the thread.

Returns:

`JNI_TRUE` - thread got a chance to run.
`JNI_FALSE` - thread did not get a chance to run.

3. Events

```
JVMPI_EVENT_ARENA_DELETE
```

Sent when a heap arena is deleted.

All objects residing in this arena are freed. An explicit `JVMPI_EVENT_OBJECT_FREE` is not sent for those objects. The profiler agent can infer all the objects currently residing in that arena by

keeping track of the object allocations in the arena and all the objects moved in and out of the arena.

This event is issued in the thread suspended mode. The profiler must not make any blocking calls such as entering a monitor or allocating from the C heap (for example, via `malloc`).

This event is always sent between a pair of `JVMPI_EVENT_GC_START` and `JVMPI_EVENT_GC_FINISH` events. The profiler agent should acquire all the locks need for processing this event in the event handler for `JVMPI_EVENT_GC_START`.

```
struct {
    jint arena_id;
} delete_arena;
```

Contents:

`arena_id` - ID of the arena being deleted.

`JVMPI_EVENT_ARENA_NEW`

Sent when a new arena for allocating objects is created.

```
struct {
    jint arena_id;
    char *arena_name;
} new_arena;
```

Contents:

`arena_id` - ID given to the arena.
`arena_name` - name of the arena.

`JVMPI_EVENT_CLASS_LOAD`

Sent when a class is loaded in the VM, or when the profiler agent requests a `JVMPI_EVENT_CLASS_LOAD` event by issuing a `RequestEvent` call. In the latter case, the `JVMPI_REQUESTED_EVENT` bit in the event type is set.

This event is issued with GC disabled. GC is re-enabled after `NotifyEvent` returns.

```
struct {
    char *class_name;
    char *source_name;
    jint num_interfaces;
    jint num_methods;
    JVMPI_Method *methods;
    jint num_static_fields;
    JVMPI_Field *statics;
    jint num_instance_fields;
    JVMPI_Field *instances;
    jobjectID class_id;
} class_load;
```

Contents:

<code>class_name</code>	- name of class being loaded.
<code>source_name</code>	- name of source file that defines the class.
<code>num_interfaces</code>	- number of interfaces implemented by this class.
<code>methods</code>	- methods defined in the class.
<code>num_static_fields</code>	- number of static fields defined in this class.
<code>statics</code>	- static fields defined in the class.
<code>num_instance_fields</code>	- number of instance fields defined in this class.
<code>instances</code>	- instance fields defined in the class.
<code>class_id</code>	- class object ID.

Note: class IDs are IDs of the class objects and are subject to change when `JVMPI_EVENT_OBJECT_MOVE` arrives.

`JVMPI_EVENT_CLASS_LOAD_HOOK`

Sent when the VM obtains a class file data, but before it constructs the in-memory representation for that class. The profiler agent can instrument the existing class file data sent by the VM to include profiling hooks.

The profiler must allocate the space for the modified class file data buffer using the memory allocation function pointer sent in this event, because the VM is responsible for freeing the new class file data buffer.

```
struct {
    unsigned char *class_data;
    jint class_data_len;
    unsigned char *new_class_data;
    jint new_class_data_len;
    void * (*malloc_f)(unsigned int);
} class_load_hook;
```

Contents:

<code>class_data</code>	- pointer to the current class file data buffer.
<code>class_data_len</code>	- length of current class file data buffer.
<code>new_class_data</code>	- pointer to the instrumented class file data buffer.
<code>new_class_data_len</code>	- length of the new class file data buffer.
<code>malloc_f</code>	- pointer to a memory allocation function.

The profiler agent must set `new_class_data` to point to the newly instrumented class file data buffer and set `new_class_data_len` to the length of that buffer before returning from `NotifyEvent`. It must set both `new_class_data` and `new_class_data_len` to the old values if it chooses not to instrument this class.

`JVMPI_EVENT_CLASS_UNLOAD`

Sent when a class is unloaded.

This event is issued with GC disabled. GC is re-enabled after `NotifyEvent` returns.

```

struct (
    jobjectID class_id;
) class_unload;

```

Contents:

class_id - class being unloaded.

JVMPI_EVENT_COMPILED_METHOD_LOAD

Sent when a method is compiled and loaded into memory.

```

struct (
    jmethodID method_id;
    void *code_addr;
    jint code_size;
    jint lineno_table_size;
    JVMPI_Lineno *lineno_table;
) compiled_method_load;

```

Contents:

method_id	- method being compiled and loaded.
code_addr	- address where compiled method code is loaded.
code_size	- size of compiled code.
lineno_table_size	- size of line number table.
lineno_table	- table mapping offset from beginning of method to the src file line number.

JVMPI_EVENT_COMPILED_METHOD_UNLOAD

Sent when a compiled method is unloaded from memory.

```

struct (
    jmethodID method_id;
) compiled_method_unload;

```

Contents:

method_id - compiled method being unloaded.

JVMPI_EVENT_DATA_DUMP_REQUEST

Sent by the VM to request the profiler agent to dump its data. This is just a hint and the profiler agent need not react to this event. This is useful for processing command line signals from users. For example, in JDK 1.2 a CTRL-Break on Win32 and a CTRL-\ on Solaris causes the VM to send this event to the profiler agent.

There is no event specific information.

JVMPI_EVENT_DATA_RESET_REQUEST

Sent by the VM to request the profiler agent to reset its data. This is just a hint and the profiler agent need not react to this event. This is useful for processing command line signals from users. For example, in JDK 1.2 a CTRL-Break on Win32 and a CTRL-\ on Solaris causes the VM to send this event to the profiler agent.

There is no event specific information.

JVMPI_EVENT_GC_FINISH

Sent when GC finishes. The profiler agent can release any locks, grabbed during GC start notification for handling object free, object move, and arena delete events. The system gets back into the multi-threaded mode after this event.

The event-specific data contains Java heap statistics.

```
struct {
    jlong used_objects;
    jlong used_object_space;
    jlong total_object_space;
} gc_info;
```

Contents:

used_objects - number of used objects on the heap.
 used_object_space - amount of space used by the objects (in bytes).
 total_object_space - total amount of object space (in bytes).

JVMPI_EVENT_GC_START

Sent when GC is about to start. The system goes into thread suspended mode after this event. To avoid deadlocks, the profiler agent should grab any locks that are needed for handling object free, object move, and arena delete events in the event handler for this event.

There is no event specific information.

JVMPI_EVENT_HEAP_DUMP

Sent when requested by the RequestEvent function. The profiler agent can specify the level of information to be dumped by passing an JVMPI_HeapDumpArg structure to RequestEvent as the second argument, with the heap_dump_level field set to the desired dump level.

The dump level values can be one of the following:

- JVMPI_DUMP_LEVEL_0
- JVMPI_DUMP_LEVEL_1
- JVMPI_DUMP_LEVEL_2

If a NULL value is passed, then the dump level is set to JVMPI_DUMP_LEVEL_2.

This event is issued with GC disabled. GC is re-enabled after NotifvEvent returns.

The event-specific data contains a snapshot of all live objects in the Java heap.


```

struct {
    int dump_level;
    char *begin;
    char *end;
    jint num_traces;
    JVMPIDumpCallTrace *traces;
} heap_dump;

```

Contents:

dump_level - the dump level specified in RequestEvent
begin - beginning of the heap dump
end - end of the heap dump
num_traces - number of stack traces in which the GC roots reside, 0 for JVMPIDUMP_LEVEL_0
traces - the stack traces in which the GC roots reside

The format of the heap dump between **begin** and **end** depends on the level of information requested. The formats are described in detail in the JVMPIDump Formats section.

JVMPIDUMP_EVENT_JNI_GLOBALREF_ALLOC

Sent when a JNI global reference is created. The event-specific data contains the JNI global reference as well as the corresponding object ID.

This event is issued with GC disabled. GC is re-enabled after NotifyEvent returns.

```

struct {
    jobjectID obj_id;
    jobject ref_id;
} jni_globalref_alloc;

```

Contents:

obj_id - object ID referred to by the global reference.
ref_id - JNI global reference.

JVMPIDUMP_EVENT_JNI_GLOBALREF_FREE

Sent when a JNI global reference is deleted. The event-specific data contains the JNI global reference that is being deleted.

```

struct {
    jobject ref_id;
} jni_globalref_free;

```

Contents:

ref_id - JNI global reference.

JVMPIDUMP_EVENT_JNI_WEAK_GLOBALREF_ALLOC

Sent when a JNI weak global reference is created. The event-specific data contains the JNI weak global reference as well as the corresponding object ID.

This event is issued with GC disabled. GC is re-enabled after `NotifyEvent` returns.

```
struct {
    jobjectID obj_id;
    jobject ref_id;
} jni_globalref_alloc;
```

Contents:

`obj_id` - object ID referred to by the weak global reference.
`ref_id` - JNI weak global reference.

JVMPI_EVENT_JNI_WEAK_GLOBALREF_FREE

Sent when a JNI weak global reference is deleted. The event-specific data contains the JNI weak global reference that is being deleted.

```
struct {
    jobject ref_id;
} jni_globalref_free;
```

Contents:

`ref_id` - JNI weak global reference.

JVMPI_EVENT_JVM_INIT_DONE

Sent by the VM when its initialization is done. It is safe to call `CreateSystemThread` only after this event is notified.

There is no event specific data.

JVMPI_EVENT_JVM_SHUT_DOWN

Sent by the VM when it is shutting down. The profiler typically responds by saving the profiling data.

There is no event specific data.

JVMPI_EVENT_METHOD_ENTRY

Sent when a method is entered. Compared with `JVMPI_EVENT_METHOD_ENTRY2`, this event does not send the `jobjectID` of the target object on which the method is invoked.

```
struct {
    jmethodID method_id;
} method;
```

Contents:

`method_id` - the method being entered.

JVMPI_EVENT_METHOD_ENTRY2

Sent when a method is entered. If the method is an instance method, the `objectID` of the target object is sent with the event. If the method is a static method, the `obj_id` field in the event is set to NULL.

This event is issued with GC disabled. GC is re-enabled after `NotifyEvent` returns.

```
struct {
    jmethodID method_id;
    objectID obj_id;
} method_entry2;
```

Contents:

`method_id` - the method being entered.
`obj_id` - the target object, NULL for static methods.

JVMPI_EVENT_METHOD_EXIT

Sent when a method is exited. The method exit may be a normal exit, or caused by an unhandled exception.

```
struct {
    jmethodID method_id;
} method;
```

Contents:

`method_id` - the method being entered.

JVMPI_EVENT_MONITOR_CONTENTED_ENTER

Sent when a thread is attempting to enter a Java monitor already acquired by another thread.

This event is issued with GC disabled. GC is re-enabled after `NotifyEvent` returns.

```
struct {
    objectID object;
} monitor;
```

Contents:

`object` - object ID associated with the monitor

JVMPI_EVENT_MONITOR_CONTENTED_ENTERED

Sent when a thread enters a Java monitor after waiting for it to be released by another thread.

This event is issued with GC disabled. GC is re-enabled after NotifyEvent returns.

```
struct {
    jobjectID object;
} monitor;
```

Contents:

object - object ID associated with the monitor

JVMPI_EVENT_MONITOR_CONTENTED_EXIT

Sent when a thread exits a Java monitor, and another thread is waiting to acquire the same monitor.

This event is issued with GC disabled. GC is re-enabled after NotifyEvent returns.

```
struct {
    jobjectID object;
} monitor;
```

Contents:

object - object ID associated with the monitor

JVMPI_EVENT_MONITOR_DUMP

Sent when requested by the RequestEvent function.

The event-specific data contains a snapshot of all the threads and monitors in the VM.

This event is issued with GC disabled. GC is re-enabled after NotifyEvent returns.

```
struct {
    char *begin;
    char *end;
    jint num_traces;
    JVMPI_CallTrace *traces;
    jint *threads_status;
} monitor_dump;
```

Contents:

begin	- start of the monitor dump buffer.
end	- end of the dump buffer
num_traces	- number of thread traces.
traces	- traces of all threads.
thread_status	- status of all threads.

The format of the monitor dump buffer is described in detail in the JVMPI Dump Formats section.

JVMPI_EVENT_MONITOR_WAIT

Sent when a thread is about to wait on an object.

This event is issued with GC disabled. GC is re-enabled after NotifyEvent returns.

```
struct {
    jobjectID object;
    jlong timeout;
} monitor_wait;
```

Contents:

object - ID of object on which the current thread is going to wait.
(NULL indicates the thread is in `Thread.sleep`.)

timeout - the number of milliseconds the thread will wait. (0 indicates waiting forever.)

JVMPI_EVENT_MONITOR_WAITED

Sent when a thread finishes waiting on an object.

This event is issued with GC disabled. GC is re-enabled after NotifyEvent returns.

```
struct {
    jobjectID object;
    jlong timeout;
} monitor_wait;
```

Contents:

object - ID of object on which the current thread waited.
(NULL indicates the thread is in `Thread.sleep`.)

timeout - the number of milliseconds the thread waited.

JVMPI_EVENT_OBJECT_ALLOC

Sent when an object is allocated, or when the profiler agent requests a **JVMPI_EVENT_OBJECT_ALLOC** event by issuing a RequestEvent call. In the latter case, the **JVMPI_REQUESTED_EVENT** bit in the event type is set.

This event is issued with GC disabled. GC is re-enabled after NotifyEvent returns.

```
struct {
    jint arena_id;
    jobjectID class_id;
    jint is_array;
    jint size;
    jobjectID obj_id;
} obj_alloc;
```

Contents:

arena_id - arena where allocated.
 class_id - class to which this object belongs, or the array element class if
 is_array is JVMPI_CLASS.
 is_array - values can be:

JVMPI_NORMAL_OBJECT	normal object
JVMPI_CLASS	array of objects
JVMPI_BOOLEAN	array of booleans
JVMPI_BYTE	array of bytes
JVMPI_CHAR	array of chars
JVMPI_SHORT	array of shorts
JVMPI_INT	array of ints
JVMPI_LONG	array of longs
JVMPI_FLOAT	array of floats
JVMPI_DOUBLE	array of doubles

size - size in number of bytes.
 obj_id - unique object ID.

JVMPI_ZVENT_OBJECT_DUMP

Sent when requested by the RequestEvent function. The objectID of the object for which a dump is being requested should be passed as the second argument to RequestEvent.

The profiler agent should request this event with GC disabled.

The event-specific data contains a snapshot of the object.

```
struct {
    jint data_len;
    char *data;
} object_dump;
```

Contents:

data_len - length of the object dump buffer
 data - beginning of the object dump

The format of the object dump buffer is described in detail in the JVMPI Dump Formats section.

JVMPI_EVENT_OBJECT_FREE

Sent when an object is freed.

This event is issued in the thread suspended mode. The profiler must not make any blocking calls such as entering a monitor or allocating from the C heap (for example, via malloc).

This event is always sent between a pair of JVMPI_EVENT_GC_START and JVMPI_EVENT_GC_FINISH events. The profiler agent should acquire all the locks need for

processing this event in the event handler for `JVMPI_EVENT_GC_START`.

```
struct {
    jobjectID obj_id;
} obj_free;
```

Contents:

`obj_id` - object being freed.

`JVMPI_EVENT_OBJECT_MOVE`

Sent when an object is moved in the heap.

This event is issued in the thread suspended mode. The profiler must not make any blocking calls such as entering a monitor or allocating from the C heap (for example, via `malloc`).

This event is always sent between a pair of `JVMPI_EVENT_GC_START` and `JVMPI_EVENT_GC_FINISH` events. The profiler agent should acquire all the locks need for processing this event in the event handler for `JVMPI_EVENT_GC_START`.

```
struct {
    jint arena_id;
    jobjectID obj_id;
    jint new_arena_id;
    jobjectID new_obj_id;
} obj_move;
```

Contents:

`arena_id` - current arena.
`obj_id` - current object ID.
`new_arena_id` - new arena.
`new_obj_id` - new object ID.

`JVMPI_EVENT_RAW_MONITOR_CONTENTED_ENTER`

Sent when a thread is attempting to enter a raw monitor already acquired by another thread.

```
struct {
    char *name;
    JVMPI_RawMonitor id;
} raw_monitor;
```

Contents:

`name` - name of the raw monitor
`id` - ID of the raw monitor

`JVMPI_EVENT_RAW_MONITOR_CONTENTED_ENTERED`

Sent when a thread enters a raw monitor after waiting for it to be released by another thread.

```

struct {
    char *name;
    JVMPI_RawMonitor id;
} raw_monitor;

```

Contents:

name - name of the raw monitor
id - ID of the raw monitor

JVMPI_EVENT_RAW_MONITOR_CONTENTED_EXIT

Sent when a thread exits a raw monitor, and another thread is waiting to acquire the same monitor.

```

struct {
    char *name;
    JVMPI_RawMonitor id;
} raw_monitor;

```

Contents:

name - name of the raw monitor
id - ID of the raw monitor

JVMPI_EVENT_THREAD_END

Sent when a thread ends in the VM.

The `env_id` field of the `JVMPI_Event` received in this event notification is the `JNIEnv` interface pointer of the thread that ended.

JVMPI_EVENT_THREAD_START

Sent when a thread is started in the VM, or when the profiler agent requests a `JVMPI_EVENT_THREAD_START` event by issuing a RequestEvent call. In the latter case, the `JVMPI_REQUESTED_EVENT` bit in the event type is set.

This event is issued with GC disabled. GC is re-enabled after NotifyEvent returns.

```

struct {
    char *thread_name;
    char *group_name;
    char *parent_name;
    jobjectID thread_id;
    JNIEnv *thread_env_id;
} thread_start;

```

Contents:

thread_name - name of thread being started.
 group_name - group to which the thread belongs.
 parent_name - name of parent.
 thread_id - thread object ID.
 thread_env_id - JNIEnv * of the thread.

Threads are associated with a JNIEnv pointer and a thread object ID. The JVMPI uses the JNIEnv pointer as the thread ID.

4. Dump Formats

4.1 Sizes and Types Used in Dump Format Descriptions

u1: 1 byte

u2: 2 bytes

u4: 4 bytes

u8: 8 bytes

ty: u1 where:

JVMPI_NORMAL_OBJECT	normal object
JVMPI_CLASS	array of objects
JVMPI_BOOLEAN	array of booleans
JVMPI_BYTE	array of bytes
JVMPI_CHAR	array of chars
JVMPI_SHORT	array of shorts
JVMPI_INT	array of ints
JVMPI_LONG	array of longs
JVMPI_FLOAT	array of floats
JVMPI_DOUBLE	array of doubles

v1: values, exact size depends on the type of value:

boolean, byte	u1
short, char	u2
int, float	u4
long, double	u8
JNIEnv *, jobjectID, and JVMPI_RawMonitor	sizeof(void *)

4.2 Heap Dump Format

The heap dump format depends on the level of information requested.

JVMPI_DUMP_LEVEL_0:

The dump consists of a sequence of records of the following format:

ty type of object
jobjectID object

JVMPI_DUMP_LEVEL_1:

The dump format is the same as that of JVMPI_DUMP_LEVEL_2, except that the following values are excluded from the dump: primitive fields in object instance dumps, primitive static fields in class dumps, and primitive array elements.

JVMPI_DUMP_LEVEL_2:

The dump consists of a sequence of records, where each record includes an 8-bit record type followed by data whose format is specific to each record type.

Record type	Record data	
JVMPI_GC_ROOT_UNKNOWN (unknown root)	jobjectID	object
JVMPI_GC_ROOT_JNI_GLOBAL (JNI global ref root)	jobjectID jobject	object JNI global reference
JVMPI_GC_ROOT_JNI_LOCAL (JNI local ref)	jobjectID JNIEnv * u4	object thread frame # in stack trace (-1 for empty)
JVMPI_GC_ROOT_JAVA_FRAME (Java stack frame)	jobjectID JNIEnv * u4	object thread frame # in stack trace (-1 for empty)
JVMPI_GC_ROOT_NATIVE_STACK (native stack)	jobjectID JNIEnv *	object thread
JVMPI_GC_ROOT_STICKY_CLASS (system class)	jobjectID	class object
JVMPI_GC_ROOT_THREAD_BLOCK (reference from thread block)	jobjectID JNIEnv *	thread object thread
JVMPI_GC_ROOT_MONITOR_USED (entered monitor)	jobjectID	object
JVMPI_GC_CLASS_DUMP (dump of a class object)	jobjectID jobjectID jobjectID jobjectID jobjectID void * void * u4 {jobjectID} * u2 {u2, ty,	class super class loader signers protection domain reserved reserved instance size (in bytes) interfaces size of constant pool constant pool index, type,

	{v1}* {v1}* value static field values
JVMPI_GC_INSTANCE_DUMP (dump of a normal object)	objectID objectID u4 {v1}* object class number of bytes that follow instance field values (class, followed by super, super's super ...)
JVMPI_GC_OBJ_ARRAY_DUMP (dump of an object array)	objectID u4 objectID {objectID}* array object number of elements element class ID (may be NULL in JDK 1.2) elements
JVMPI_GC_PRIM_ARRAY_DUMP (dump of a primitive array)	objectID u4 ty {v1}* array object number of elements element type elements

4.3 Object Dump Format

The dump buffer consists of a single record which includes an 8-bit record type, followed by data specific to the record type. The record type can be one of the following:

- JVMPI_GC_CLASS_DUMP
- JVMPI_GC_INSTANCE_DUMP
- JVMPI_GC_OBJ_ARRAY_DUMP
- JVMPI_GC_PRIM_ARRAY_DUMP

The format of the data for each record type is the same as described above in the heap dump format section. The level of information is the same as JVMPI_DUMP_LEVEL_2, with all of the following values included: primitive fields in object instance dumps, primitive static fields in class dumps, and primitive arrays elements.

4.4 Monitor Dump Format

The dump buffer consists of a sequence of records, where each record includes an 8-bit record type followed by data whose format is specific to each record type.

Record type	Record data
JVMPI_MONITOR_JAVA (Java monitor)	objectID JNIEnv * u4 u4 {JNIEnv *} * u4 {JNIEnv *} * object ID owner thread entry count number of threads waiting to enter threads waiting to enter number of threads waiting to be notified threads waiting to be notified

JVMPI_MONITOR_RAW (Raw monitor)	char *	raw monitor name
	JVMPI_RawMonitor	raw monitor ID
	JNIEnv *	owner thread
	u4	entry count
	u4	number of threads waiting to enter
	[JNIEnv *]	threads waiting to enter
	u4	number of threads waiting to be notified
[JNIEnv *]	threads waiting to be notified	

5. Data Types

Characters are encoded using the UTF-8 encoding as documented in the Java virtual machine specification.

jobjectID

An opaque pointer representing an object ID.

```
struct _jobjectID;
typedef struct _jobjectID * jobjectID;
```

JVMPI_CallFrame

A method being executed.

```
typedef struct {
    jint lineno;
    jmethodID method_id;
} JVMPI_CallFrame;
```

Fields:

lineno - line number in the source file.
 method_id - method being executed.

JVMPI_CallTrace

A call trace of method execution.

```
typedef struct {
    JNIEnv *env_id;
    jint num_frames;
    JVMPI_CallFrame *frames;
} JVMPI_CallTrace;
```

Fields:

env_id - ID of thread which executed this trace.
num_frames - number of frames in the trace.
frames - the JVMPI CallFrames that make up this trace. Callee followed by callers.

JVMPI_Field

A field defined in a class.

```
typedef struct {
    char *field_name;
    char *field_signature;
} JVMPI_Field;
```

Fields:

field_name - name of field
field_signature - signature of field

JVMPI_HeapDumpArg

Additional info for requesting heap dumps.

```
typedef struct {
    jint heap_dump_level;
} JVMPI_HeapDumpArg;
```

Fields:

heap_dump_level - level of heap dump information, values can be:

```

    JVMPI_DUMP_LEVEL_0
    JVMPI_DUMP_LEVEL_1
    JVMPI_DUMP_LEVEL_2
  
```

JVMPI_Lineno

A mapping between source line number and offset from the beginning of a compiled method.

```
typedef struct {
    jint offset;
    jint lineno;
} JVMPI_Lineno;
```

Fields:

offset - offset from beginning of method
lineno - lineno from beginning of source file

JVMPI_Method

A method defined in a class:

```
typedef struct {
    char *method_name;
    char *method_signature;
    jint start_lineno;
    jint end_lineno;
    jmethodID method_id;
} JVMPI_Method;
```

Fields:

```
method_name      - name of method
method_signature - signature of method
start_lineno     - start line number in the source file
end_lineno       - end line number in the source file
method_id        - ID given to this method
```

JVMPI_RawMonitor

An opaque pointer representing a raw monitor.

```
struct _JVMPI_RawMonitor;
typedef struct _JVMPI_RawMonitor * JVMPI_RawMonitor;
```

6. Notes on JDK1.2 Implementation Limitations

- JVMPI_EVENT_OBJECT_ALLOC events for object arrays are issued with unknown element class IDs (i.e., the class_id field is always NULL).
- On Win32 the following events are not yet supported in the presence of the JIT compiler:
 - JVMPI_EVENT_METHOD_ENTRY,
 - JVMPI_EVENT_METHOD_ENTRY2,
 - JVMPI_EVENT_METHOD_EXIT,
 - JVMPI_EVENT_COMPILED_METHOD_LOAD, and
 - JVMPI_EVENT_COMPILED_METHOD_UNLOAD,
- SuspendThread must be called with the GC is disabled. GC must remain disabled until all threads have been resumed.
- The thread start event for the main thread (first thread the VM creates) may arrive after some other events that refer to its JNIEnv interface pointer.
- JVMPI_EVENT_ARENA_NEW and JVMPI_EVENT_ARENA_DELETE events are never issued. Arena IDs in other events are always set to 1.

Last modified: Wed Nov 11 14:14:44 PST 1998



PATENT
Attorney Docket No. 06502.0523-00

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of:

SHENG LIANG et al.

Application No.: 09/856,779

Filed: October 3, 2001

For: A METHOD FOR ENABLING
COMPREHENSIVE PROFILING OF
GARBAGE-COLLECTED MEMORY
SYSTEMS

) Group Art Unit: 2187

) Examiner: Brian R. Peugh

) Confirmation No.: 3704

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Sir:

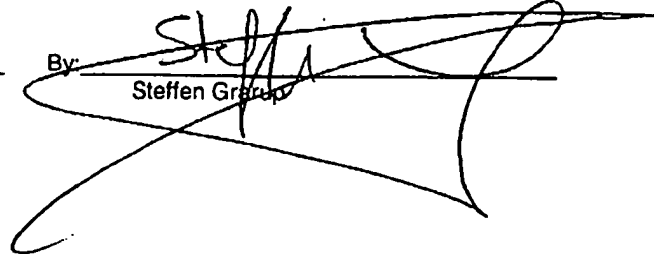
DECLARATION UNDER 37 C.F.R. § 1.132

I, Steffen Grarup, do hereby make the following declaration:

1. I have reviewed the Declaration Under 37 C.F.R. § 1.132 executed by Sheng Liang on October 10, 2006. To the best of my knowledge, all statements in Mr. Liang's Declaration is true.
2. Specifically, Mr. Liang authored the document, "Java Virtual Machine Profiler Interface (JVMPI)," November 11, 1998 (Appendix A in Mr. Liang's Declaration).
3. To the extent that Appendix A in Mr. Liang's Declaration describes the invention(s) in the above-referenced patent application, Mr. Liang and I are the inventors.

I further declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true, and further, that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application or any patent issuing thereon.

Dated: 6/11/2007

By: 
Steffen Grarup